Algorithms and Data Structures 2
CS 1501

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - **Assignment 2: tonight @ 11:59 pm**
    - **NO LATE DEADLINE**
  - Lab 7: tonight @ 11:59 pm
  - Homework 8: this Friday @ 11:59 pm
- Autograder issues
  - OutOfMemoryError because of a cycle in the trie

Previous lecture

- Minimum Spanning Tree (MST) problem
  - Prim's MST algorithm
    - naive implementation (Theta(v^3))
    - Using Best Edges array

This Lecture

- Minimum Spanning Tree (MST) problem
  - Prim’s MST algorithm
    - running time analysis of the Best Edges implementation
    - an implementation that uses a heap
  - Kruskal’s MST algorithm

Muddiest Points

- **How does the summation of (i *(v - i)) equal Theta of (largest term* * number of terms) instead of O?**
- $\sum_{i=1}^{v-1} i(v-i)$ is the running time for the naive implementation of Prim's MST algorithm
- Example: v = 10
- $\sum_{i=1}^{9} i(10 - i) = 9 + 16 + 21 + 24 + \mathbf{25} + 24 + 21 + 16 + 9$
- $\sum_{i=1}^{v-1} i(v-i) = \mathrm{v}\sum_{i=1}^{v-1} i \, - \sum_{i=1}^{v-1} i$
- Although largest term * number of terms is an upper bound on the sum of an arithmetic series, it is within a constant factor

Muddiest Points

- **can we see another example of enhanced Prim's?**
- Sure!

Muddiest Points

- **what is the runtime when using best edge?**
- We will see that today.

Runtime of the Best Edges Implementation

- For every vertex we add to T, we'll need to check all of its neighbors to update their best edges as needed
  - Let's assume we use an **adjacency matrix**:
    - Takes Θ(v) to check the neighbors of a given vertex
    - Time to update parent/best edge arrays?
      - Θ(1)
    - Time to pick next vertex?
      - Θ(v)
    - Total: v*2 Θ(v) = Θ(v^2)

OK, so what's our runtime?

- For every vertex we add to T, we'll need to check all of its neighbors to update their best edges as needed
  - Let's assume we use **adjacency lists**
    - Takes Θ(d) to check the neighbors of a given vertex
    - Time to update parent/best edge arrays?
      - Θ(1)
    - Time to pick next vertex?
      - Θ(v)
    - Total: $v*\Theta(v + d) = \Theta(v^2)$

Prim’s MST Algorithm

- seen, parent, and BestEdge are arrays of size v
- Initialize seen to false, parent to -1, and BestEdge to infinity
- BestEdge[start] = 0
- for i = 0 to v-1
  - Find a vertex w with seen[w] = false and BestEdge[w] is the minimum over all unseen vertices
  - seen[w] = 1
  - for each neighbor x of w
    - if(BestEdge[x] > edge weight of edge (w, x)
      - BestEdge[x] = edge weight of (w, x)
      - parent[x] = w
- The parent array represents the found MST

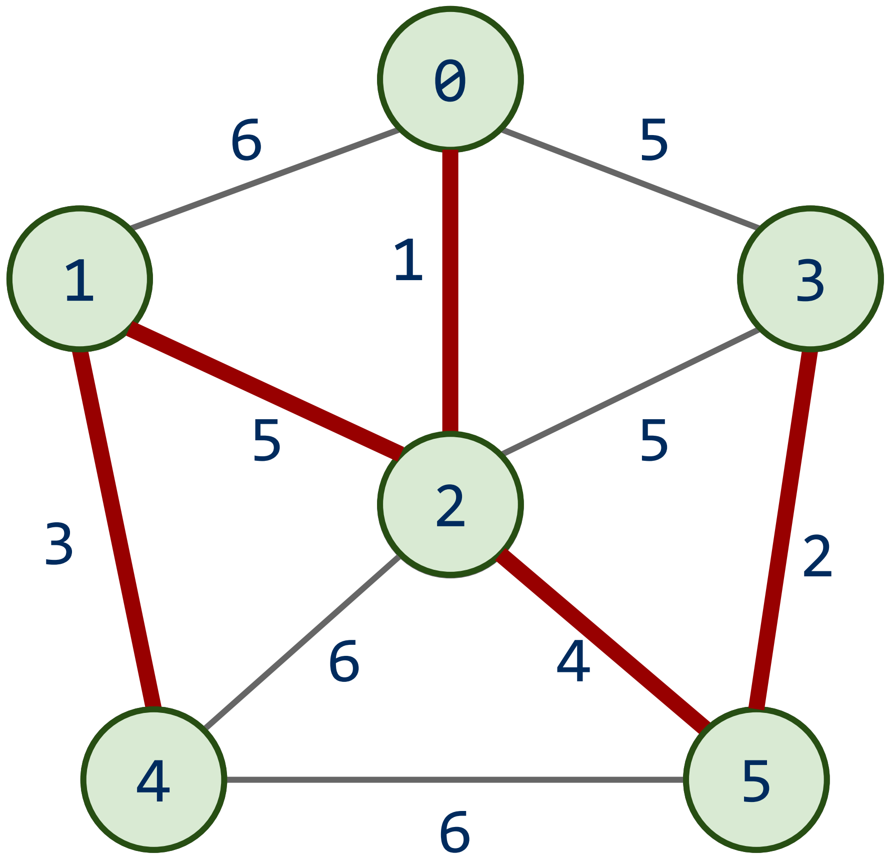
Prim's MST Algorithm

- seen, parent, and BestEdge are arrays of size v
- Initialize seen to false, parent to -1, and BestEdge to infinity
- BestEdge[start] = 0
- for i = 0 to v-1
  - **Find a vertex w with seen[w] = false and BestEdge[w] is the minimum over all unseen vertices**
  - seen[w] = 1
  - for each neighbor x of w
    - if(BestEdge[x] > edge weight of edge (w, x)
      - BestEdge[x] = edge weight of (w, x)
      - parent[x] = w
- The parent array represents the found MST

What about a faster way to pick the best edge?

- Sounds like a job for a priority queue!
  - Priority queues can remove the min value stored in them in Θ(lg n)
    - Also Θ(lg n) to add to the priority queue
- What does our algorithm look like now?
  - Visit a vertex
  - Add edges coming out of it to a PQ
  - While there are unvisited vertices, pop from the PQ for the next vertex to visit and repeat

Prim's with a priority queue



PQ:

1: (0, 2)
2: (5, 3)
3: (1, 4)
4: (2, 5)
5: (2, 3)
5: (0, 3)
5: (2, 1)
6: (0, 1)
6: (2, 4)
6: (5, 4)

Runtime using a priority queue

- Have to insert all e edges into the priority queue
  - In the worst case, we'll also have to remove all e edges
- So we have:
  - $e * \Theta(\lg e) + e * \Theta(\lg e)$
  - $= \Theta(2 * e \lg e)$
  - $= \Theta(e \lg e)$
- This algorithm is known as *lazy Prim's*

Do we really need to maintain e items in the PQ?

- I suppose we could not be so lazy
- Just like with the best edge array implementation, we only need the best edge for each vertex
  - PQ will need to be indexable to update the best edge
- This is the idea of *eager Prim's*
  - Runtime is $\Theta(e \lg v)$

Eager Prim's Runtime

- v inserts
  - v log v
- e updates
  - e log v
- v removeMin
  - v log v
- Total: (e+v) log v
- Assuming connected graph
  - e >= v – 1
- e+v = Theta(e)
- Total runtime = e log v

Comparison of Prim's implementations

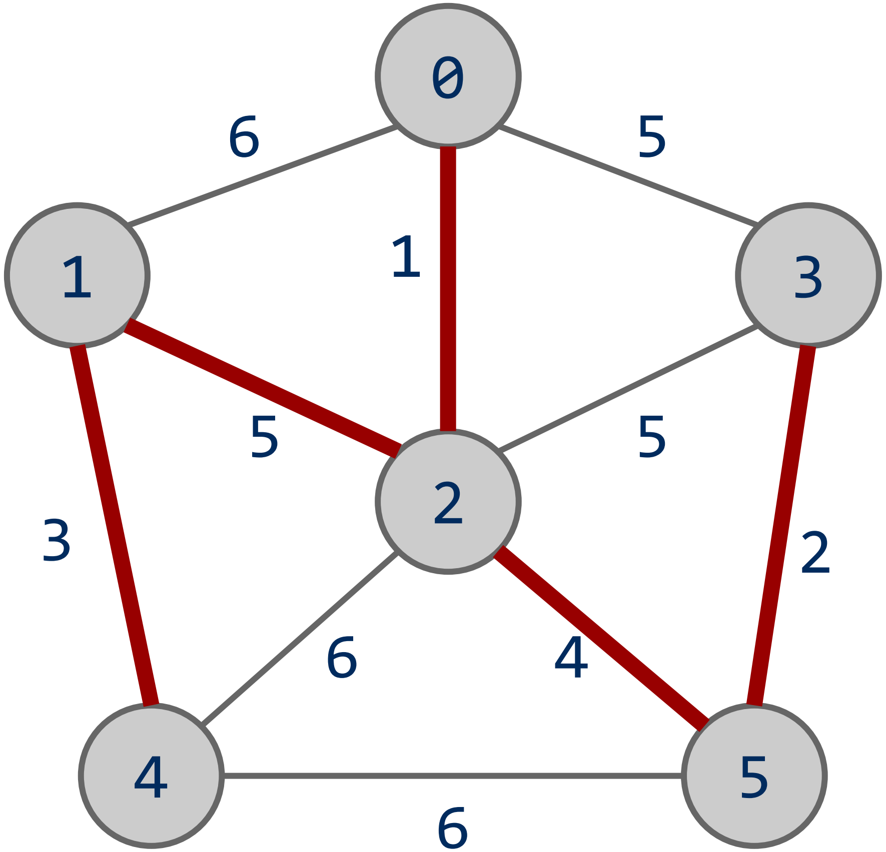
- Parent/Best Edge array Prim's
  - Runtime: $\Theta(v^2)$
  - Space: $\Theta(v)$
- Lazy Prim's
  - Runtime: $\Theta(e \lg e)$
  - Space: $\Theta(e)$
  - Requires a PQ
- Eager Prim's
  - Runtime: $\Theta(e \lg v)$
  - Space: $\Theta(v)$
  - Requires an indexable PQ

How do these compare?

Another MST algorithm

- Kruskal's MST:
    - Insert all edges into a PQ
    - Grab the min edge from the PQ that does not create a cycle in the MST
    - Remove it from the PQ and add it to the MST

Kruskal's example



PQ:

1: (0, 2)

2: (3, 5)

3: (1, 4)

4: (2, 5)

5: (2, 3)

5: (0, 3)

5: (1, 2)

6: (0, 1)

6: (2, 4)

6: (4, 5)

Kruskal’s runtime

- Instead of building up the MST starting from a single vertex, we build it up using edges all over the graph
- How do we efficiently implement cycle detection?
  - BFS/DFS
    - v + e
  - Union/Find data structure
    - log v

Kruskal’s Runtime

- e iterations
  - removeMin
    - log e
  - Cycle detection
    - v + e using DFS/BFS
    - log v using Union/Find
- Total: e log e
- Assuming connected graph
  - $v - 1 <= e <= v^2$
  - log v <= log e <= 2 log v
  - log e = Theta(log v)
- Total runtime: e log v
- Same as Prim’s

Problem of the Day 1: Weighted Shortest Path

- Input:
  - A road network
    - Road segments and intersections
    - Road segments are labeled by travel time
      - From length and maximum speed
      - How do we get max speed?
  - Starting address and destination address
- Output:
  - A shortest path from source to destination

Weighted shortest path

- Dijkstra's algorithm:
  - Set a distance value of MAX_INT for all vertices but start
  - Set cur = start
  - While destination is not visited:
    - For each unvisited neighbor of cur:
      - Compute tentative distance from start to the unvisited neighbor through cur
      - Update any vertices for which a lesser distance is computed
    - Mark cur as visited
    - Let cur be the unvisited vertex with the smallest tentative distance from start

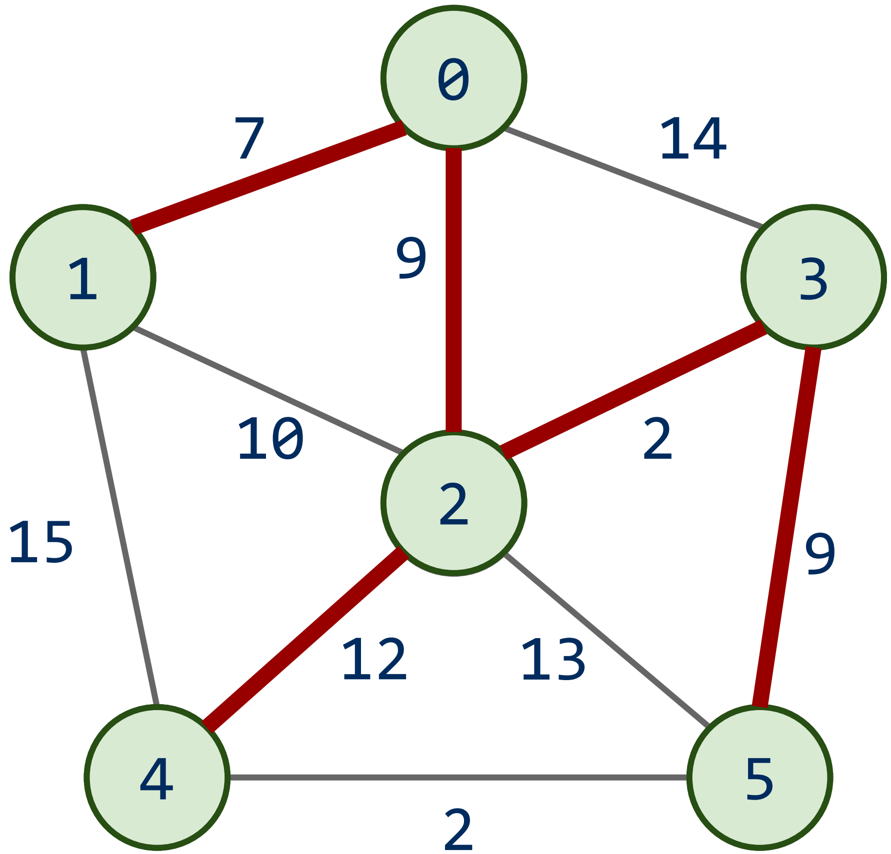
Tentative Distance

- tentative distance from the start vertex to an unvisited neighbor

  through the current vertex =

  distance[cur] + edge weight between cur and neighbor

Dijkstra's example



	Distance	Parent
0	0	--
1	7	0
2	9	0
3	11	2
4	21	2
5	20	3

Analysis of Dijkstra's algorithm

- How to implement?
  - Best path/parent array?
    - Runtime?
  - PQ?
    - Turns out to be very similar to Eager Prims
      - Storing paths instead of edges
    - Runtime?